Network Coding Over The 232

5 Prime Field

Pedersen, Morten Videbæk; Heide, Janus; Vingelmann, Peter; Fitzek, Frank

Published in:
Communications (ICC), 2013 IEEE International Conference on

DOI (link to publication from Publisher):
10.1109/ICC.2013.6654986

Publication date:
2013

Document Version
Accepted author manuscript, peer reviewed version

Network Coding Over The $2^{32} - 5$ Prime Field

Morten Videbæk Pedersen[†], Janus Heide[†], Péter Vingelmann[*†], and Frank H. P. Fitzek[†]
[†]Department of Electronic Systems, Faculty of Engineering and Science, Aalborg University, Denmark
[*]Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Hungary

***Abstract*—Creating efficient finite field implementations has been an active research topic for several decades. Many applications in areas such as cryptography, signal processing, erasure coding and now also network coding depend on this research to deliver satisfactory performance. In this paper we investigate the use of prime fields with a field size of $2^{32} - 5$, as this allows implementations which combines high field sizes and low complexity. First we introduce the algorithms needed to apply prime field arithmetics to arbitrary binary data. After this we present the initial throughput measurements from a benchmark application written in C++. These results are finally compared to different binary and binary extension field implementations. The results show that the prime field implementation offers a large field size while maintaining a very good performance. We believe that using prime fields will be useful in many network coding applications where large field sizes are required.**

I. Introduction

With its introduction by Ahlswede et al. in 2000 [1], Network Coding (NC) has undergone a tremendous evolution, from simple XOR type coding schemes to newer developments such as Random Linear Network Coding (RLNC). NC has shown its potential in a variety of application fields covering sensor networks, satellite networks and peer-to-peer (P2P) networks as a short list of examples. The core idea behind NC schemes is to change the way packets are processed in the network. NC breaks with the traditional paradigm in packet switched networks often referred to as *store-and-forward*. In this type of network, nodes on the intermediate path of a packet flow simply receives and forwards the incoming packets. In NC intermediate nodes in the network may choose to recode packets before forwarding them. Due to this, networks utilizing NC are often referred to as *compute-and-forward*. As shown with the famous butterfly example [1], this approach can lead to significant throughput gains. The arithmetic operations performed by the NC nodes in a network are defined within a branch of mathematics known as finite fields or Galois fields. Finite fields define all the commonly used arithmetic operations i.e. addition, subtraction, multiplication and division. These arithmetics operations are used in NC when performing the three core operations namely: encoding, recoding and decoding. Efficient implementations of finite field arithmetics are therefore an important prerequisite for any efficient NC implementation. See [2] for a thorough introduction to the theory of finite fields.

In both software and hardware, finite fields may be implemented in a number of different ways, and in general one cannot point out a single superior implementation covering all possible use-cases [3]. Different applications will often have different requirements. The finite field implementation presented in this paper addresses two requirements commonly seen in NC applications, namely high field sizes and low algorithmic memory consumption.

A. Field Size

Depending on the network topology, a large field size may be required to efficiently realize the communication. As stated by the main theorem of NC given in [4].

Theorem 1 (Main Theorem in Network Coding): Consider a directed acyclic graph $G = (V, E)$ with unit capacity edges, h unit rate sources located on the same vertex of the graph and N receivers. Assume that the value of the min-cut to each receiver is h. Then there exists a multicast transmission scheme over a large enough finite field $\mathbb{F}_q$, in which intermediate network nodes linearly combine their incoming information symbols over $\mathbb{F}_q$, that delivers the information from the sources simultaneously to each receiver at a rate equal to h.

Even in very simple network topologies, choosing a too small field size can reduce the effectiveness of the coding by introducing an excess of linearly dependent packets [5]. However, note that increasing the field size in RLNC schemes also increases the overhead added to each encoded symbol through the encoding vector. It is therefore undesirable to increase the field size more than necessary, an overview of this trade-off is provided in [6].

B. Memory Consumption

On memory constrained devices, algorithms with a limited memory consumption may be a requirement. Many finite field implementations used in NC applications rely on different variants of lookup tables to provide efficient arithmetic operations. On constrained devices lookup tables may be undesirable as they occupy valuable space in memory (e.g. on a sensor board) and can cause severe performance issues due to the typically limited Central Processing Unit (CPU) cache size [3].

In this paper we will introduce the initial implementation results utilizing a new scheme based on results from Optimal Extension Fields (OEFs) and a sub-field mapping algorithm developed by Crowley et al. [7]. This scheme allows for both a high field size and a very low memory consumption.

The remainder of the paper is organized as follows. In Section II OEFs are described. Section III introduces the algorithm for fast modulo reduction. Section IV introduces the binary sub-field mapping algorithm. Section V presents the implementation and measurements of a search algorithm

required by the sub-field mapping algorithm. Section VI presents an overview of the finite field operations performed by NC implementations. Section VII presents the obtained results from an initial implementation. The final conclusions are drawn in Section VIII.

II. Optimal Extension Fields

OEFs were introduced in [8] for use in public-key cryptographic systems. OEFs differ from the traditional binary field implementations by changing the characteristic of the field. The general definition of a field is given as $\mathbb{F}_{p^m}$, for which p is a prime number also called the field characteristic and m denotes the extension used. Efficient finite field arithmetics in an OEF are achieved by choosing a characteristic (i.e. a prime p) close to the word size of the underlying hardware processor e.g. 8, 16, 32 or 64 bits and creating field extensions using irreducible polynomials of a special form. This approach differs from traditional implementations where a binary extension field is used, i.e. fields with characteristic $p = 2$. As an example, a concrete implementation for a 32-bit CPU may use a binary extension field such as $\mathbb{F}_{2^{32}}$, whereas an OEF implementation could use $\mathbb{F}_{4294967291^m}$, where 4294967291 is the prime number $2^{32}-5$. Although Bailey et al. envisioned OEFs to be used in cryptographic systems, we may use their results to also create efficient prime field implementations for NC applications.

Before moving on, let us give the definitions describing an OEF as described in [8]:

- **Definition 1.** A pseudo-Mersenne prime is a prime number of the form $2^n - c$, where $\log_2(c) \leq \frac{1}{2}n$
- **Definition 2.** An Optimal Extension Field is a finite field $\mathbb{F}_{p^m}$ such that:
  1) p is a pseudo-Mersenne prime.
  2) An irreducible bionomial $p(x) = x^m - \omega$ exists over $\mathbb{F}_p$.

  We observe that there are two special cases of OEF which yield additional arithmetic advantages, which we will call Type I and Type II.
- **Definition 3.** A Type I OEF has $p = 2^n - 1$. A Type I OEF allows for subfield modular reduction with very low complexity as we will show later.
- **Definition 4.** A Type II OEF has an irreducible binomial $x^m - 2$. A Type II OEF allows speedups in extension field modular reduction.

In the following we will show how the ideas of OEFs can be utilized in a NC context. Since most NC systems do not require the same large fields as certain cryptographic systems, we will not discuss the construction of extension fields, but focus on the OEF sub-field which is given as a regular prime field $\mathbb{F}_p$, where p is a pseudo-Mersenne prime. For this reason we will refer to this type of finite field as an Optimal Prime Field (OPF). To implement this in practice, two open questions must be addressed: How to implement fast modulo reduction in the chosen prime field $p = 2^n - c$ and how to ensure that arbitrary input data can be represented within the chosen field.

III. Fast Sub-Field Modulo Reduction

This result originally stems from [9] and was later utilized in OEFs. In a finite field, the modulo operation is used to ensure that all arithmetic operations performed in the field remain "closed", i.e. adding, subtracting, multiplying or dividing two field elements must result in an element also in the field.

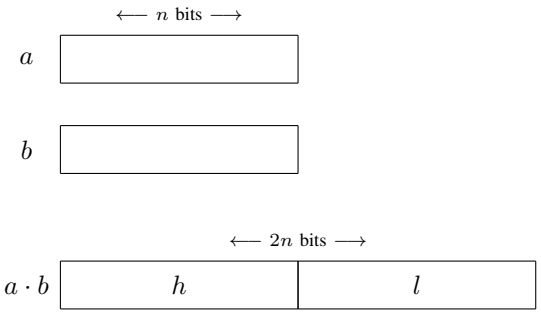
To create an efficient prime field implementation, an efficient way of implementing the modulo reduction is therefore also needed. Given two n-bit integers $a,\ b \in \mathbb{F}_p$ we may handle the addition and subtraction with reduction modulo p as:

$$a + b = \begin{cases} a + b & \text{if } a + b < p \\ a + b - p & \text{if } a + b \geq p \end{cases} \tag{1}$$

$$a - b = \begin{cases} a - b & \text{if } a - b \geq 0 \\ a - b + p & \text{if } a - b < 0 \end{cases} \tag{2}$$

For multiplication and division the process is more involved. Division can be considered a multiplication with the inverse element, therefore it will only be necessary to perform the modulo reduction after multiplication. The inverse of a field element can be calculated using the extended Euclidean Algorithm, see [10]. As with addition and subtraction, we again consider two n-bit integers $a,\ b \in \mathbb{F}_p$. Recall from "Definition 1" that p is a pseudo-Mersenne prime of the form $p = 2^n - c$, where $\log_2(c) \leq \frac{1}{2}n$. The goal is to calculate the modulo reduction of the product of two n-bit integers using only additions, shifts and multiplications. Although this may sound more complicated than calculating the modulo reduction using an integer division it will often be faster in practice due to the high latency of the integer division instruction on the CPU [11]. In the following the idea behind the modulo reduction algorithm is presented.

First choose a pseudo-Mersenne prime of the form $p = 2^n - c$, where $\log_2(c) \leq \frac{1}{2}n$. Calculate the product of the two n-bit integers a and b:



The result $a \cdot b$ can be represented as a n-bit high-half h and a n-bit low-half l. From this, it can be seen that the product $a \cdot b$ can be rewritten as:

$$a \cdot b = h \cdot 2^n + l \mod p \tag{3}$$

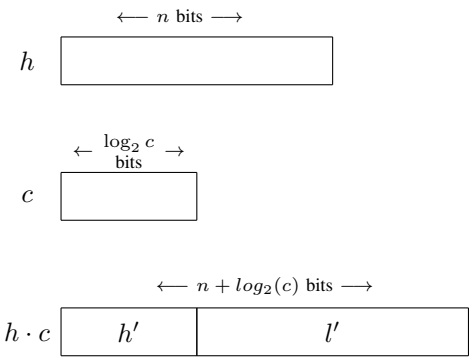
Rewriting the product as a sum of the low- and high-half, we see that the factor 2^n can be reduced using the chosen prime p, where $p = 2^n - c$:

$$2^n \equiv c \mod 2^n - c \tag{4}$$

Substituting this result into Equation (3) yields:

$$a \cdot b = h \cdot c + l \mod p \tag{5}$$

Graphically we may represent the product $h \cdot c$ as:



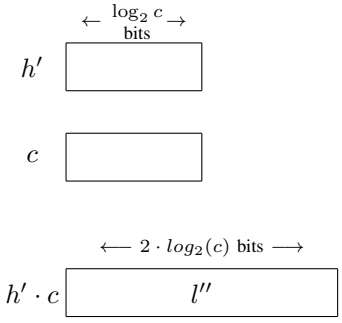
Again we represent the result as a high-half added to the low-half:

$$h \cdot c = h' \cdot 2^n + l' \mod p \tag{6}$$

Repeating the reduction step using Equation (4) we may rewrite the product:

$$h \cdot c = h' \cdot c + l' \mod p \tag{7}$$

The product $h' \cdot c$ can now be represented as:



Where l'' can be represented using $2 \cdot log_2(c) \leq n$ bits. This is possible since we have chosen c as $\log_2(c) \leq \frac{1}{2}n$. This represents the final step in the algorithm as no further high-half bits are produced. We may note that due to our choice of c this is guaranteed to happen. By combining the results from Equation (5) and (7), we may rewrite the multiplication as a sum of three $\leq n$ bit integers:

$$a \cdot b = l + l' + l'' \mod p \tag{8}$$

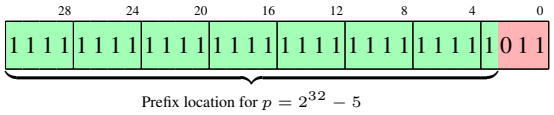
As shown for addition in Equation (1), we only have to ensure that the sum is below p by subtracting p if necessary.

IV. Sub-Field Data Mapping

As mentioned, one of the goals of OEFs is to match the underlying processor word size as closely as possible. However, in practice this means that the finite field elements cannot be exactly represented by any common data types i.e. 8, 16, 32 or 64 bits, due to the fact that a prime must be used. As an example, using a prime field with characteristic $p = 2^{32} - 5$, we see that it does not allow the binary values from `0xfffffffb` to `0xffffffff`. This would create problems for most NC applications, where the data being processed cannot be guaranteed to be within a certain binary range. Consequently, in this example we require an encoding/decoding scheme for mapping arbitrary 32-bit data values into values in the $[0, 2^{32} - 5)$ range. Using the approach presented by Crowley et al. [7], this goal may be achieved with limited computational overhead. The algorithm may be summarized in the following steps:

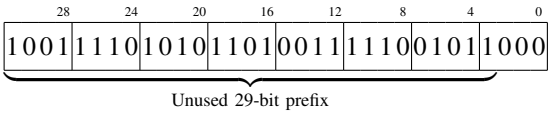
1) Choose a data type of n bits to match the processor word size.
2) Select a pseudo-Mersenne prime of the form $p = 2^n - c$.
3) Partition the input data into blocks of maximum $2^t - 1$ data words, where $t \leq \lceil n - log_2(c) \rceil$.
4) For each block, search the data to find a t-bit prefix not present in the data. Note, that since we have a maximum block size of $2^t - 1$, this prefix is guaranteed to exist.
5) Negate the t bit prefix and XOR it with all n-bit words in the data block. This will ensure that all data values are representable in the chosen prime field.
6) When no more finite field operations are required, e.g. after the data block has been transmitted and decoded, reverse the prefix mapping by performing the XOR again with the negated prefix on the data.

The outlined algorithm will ensure that all values larger than the selected prime will be mapped to a value representable with in the prime field. As an example of how the binary mapping works, consider the binary representation of the pseudo-Mersenne prime $p = 2^{32} - 5$, one way to make sure that all input values are below the prime is to ensure that at least a single zero will appear in the top 29 bits, denoted as the *prefix*:

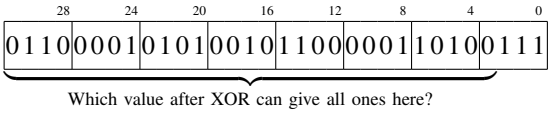


The binary sub-field mapping algorithm achieves this by partitioning the input data into blocks with a maximum of $2^{29} - 1$ data words. Corresponding to a maximum block size of approximately 2.1 GB. This block size guarantees that a 29 bit prefix, s, may be found which does not appear in any of the data words. Using the prefix s, a binary mask can be constructed and XOR'ed with the remaining data block values. The binary mask is simply the negated version of the

prefix. To illustrate how this works, assume we have found a prefix which does not appear anywhere in our data block:



Negate this prefix:



Now notice that the only value for which the XOR with the negated prefix can produce all binary ones is the prefix itself - which does not exist in the data block. XOR'ing with all other input values with the negated prefix is guaranteed to have at least a single zero bit in the top t bits. Therefore, all data values in the block will be representable within the prime field. Note, that XOR'ing will not "corrupt" the input data as it is a fully reversible operation.

In general the length of a data block will determine the number of bits necessary to ensure that an unused prefix can be found. This relationship between the block length b in 32-bit data words and prefix length t in bits is given by Equation 9.

$$t = \lceil \log_2(b+1) \rceil \text{ where } t \leq \lceil n - log_2(c) \rceil \tag{9}$$

As an example for a block length of 15 data words it is given that at least one 4-bit prefix must exist, which does not appear anywhere in the data.

V. Prefix Search

One drawback of this solution is that finding the prefix will require processing of the entire data block. This is however only necessary once, and may therefore be precomputed and attached to a storage object as meta-data. However, for steaming applications data is produced on-the-fly and therefore the prefix search has to be executed as the blocks are made available. In the following we will therefore investigate the added processing overhead of the prefix search. Searching for the prefix may be accomplished in several ways. Here we consider a *k-pass binary search*.

The k-pass binary search performs k iterations over the data block to find an unused prefix. Utilizing more than a single iteration reduces the memory consumption but is expected to increase computational cost of the algorithm. The general principle of the binary search algorithm is that given a t bit prefix we may inspect only a subset j of the bits, where $j < t$ to determine first j-bits of the unused prefix. This can be achieved by noticing that for every j-bit prefix there can be at most 2^{t-j} bit patterns. It is given that if the value of a counter is less than 2^{t-j}, then there must exist a t-bit prefix with the top j bits which does not appear in the data. Recall this is guaranteed to happen since we allow at most $2^t - 1$ data words in a block. Equation 10 gives the relationship between k, t and j.

$$j = \left\lceil \frac{t}{k} \right\rceil \quad [bits] \tag{10}$$

Utilizing the counters for each prefix the algorithm chooses a j-bit prefix for which it knows that an unused t-bit prefix must exist. In case of several candidates any can be chosen. Using the first j-bit prefix as a filter it continues by counting the next j-bit prefixes. This process continues until a counter contains the value 0, when this happens the concatenation of the chosen j bit prefices will constitute the unused t-bit prefix. The value k determines the maximum number of iterations needed. Increasing k reduces the memory requirements as less counters have to be stored. The total memory requirements can be calculated as the number of counters needed, multiplied by the size of each counter (in our implementation we used `uint32_t` which is 4 bytes), as given in Equation 11.

$$binary\_search_{memory} = 2^j \cdot 4 \quad [B] \tag{11}$$

Worthwhile noticing is that even for a generation size of 2048 it is possible to run the prefix search using less than 1000 bytes of memory using the binary search $k = 4$.

A. Prefix Search Performance

In this section we benchmark the presented prefix search algorithms. The benchmark measures the time required for the algorithm to find a missing prefix in a block of random data. In order to get an impression of the performance of the algorithms the benchmark have been conducted on the following device:

TABLE I: Specifications of the device used for benchmarking

Device	Desktop PC
OS	XUbuntu 12.04
CPU	Intel(R) Core(TM) i7 920 @ 2.67 GHz
Cache	L1 128 KiB, L2 1 MiB, L3 8 MiB
Memory	6 GB DDR3 1066 MHz

In Table II we show the results for the fastest algorithm Binary search $k = 2$, these result signify the overhead which would be added per block due to the prefix search. Depending on the type of application this could be problematic, e.g. for live streaming applications the prefix must be found "on-the-fly" as data is being produced, whereas for static objects such as files the prefix may be found in advance and attached to the data as meta-information.

TABLE II: Prefix search time [ms] for Binary search $k = 2$

Generation size	16	32	64	128	256	512	1024
Time [ms]	0.19	0.23	0.27	0.54	1.09	2.30	5.99

VI. Network Coding Arithmetics

In order to quantify the potential performance of different finite field implementations it is necessary to understand the statistics of the arithmetic operations performed in NC algorithms. For a Gaussian Elimination decoder and a generation size g, packet size of f finite field elements and an encoding vector of v finite field elements we expect the upper bound for the operations as shown in Table III.

TABLE III: Upper bounds for operations of a Gaussian Elimination decoder for a single generation

Operation	`dest[i] = dest[i] - (constant * src[i])`
Upper bound	$O((g^2 - g) \cdot (v + f))$
Operation	`dest[i] = dest[i] * constant`
Upper bound	$O(g \cdot (v + f))$
Operation	`invert(value)`
Upper bound	$O(g)$

Notice, that the length of the encoding vector is always $v = g$, however to make it clearer where the operations are coming from we will keep using v to denote its length. For the encoding algorithms we have the upper bound expression shown in Table IV.

TABLE IV: Upper bound for operations of a standard RLNC encoder for a single generation

Operation	`dest[i] = dest[i] + (constant * src[i])`
Upper bound	$O(g^2 \cdot f)$

To confirm these expressions the following results were obtained by instrumenting one standard RLNC encoding algorithm and a Gaussian Elimination based RLNC decoding algorithm (source code available here [12]). Field coefficients used for the encoding were drawn uniformly.

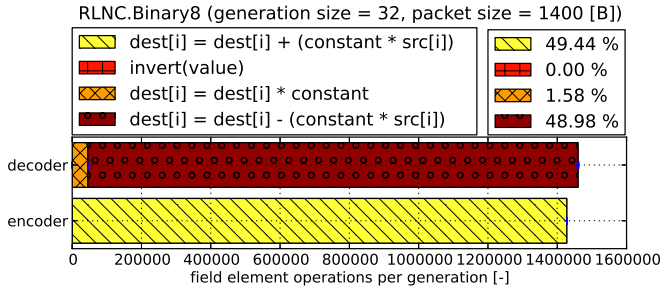


Fig. 1: Number of finite field operations needed during encoding and decoding of a single generation. The field used corresponds to the binary extension field $\mathbb{F}_{2^8}$, which means that every field element corresponds to 1 byte.

These results confirm the expectation from the upper bound expressions given in Table III and IV.

VII. Finite Field Arithmetics

This section presents the benchmark results of the suggested finite field algorithm.

Based on the results from the previous section the majority of operations performed are the two compound operations: `dest[i] = dest[i] + (constant * src[i])` and `dest[i] = dest[i] - (constant * src[i])`. These operations are therefore of key interest when comparing different finite field implementations for NC algorithms. In the following we have tested a number of different finite field implementations (source code available here [13]):

- *SimpleOnline*$\{8, 16\}$: This algorithm computes the result on-the-fly in $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{16}}$ using an iterative algorithm, without any precomputed lookup table.
- *OptimalPrime2325*: This corresponds to the algorithm presented in this paper. Using the prime field $\mathbb{F}_{4294967291}$, where $p = 2^{32} - 5 = 4294967291$
- *FullTable8*: This algorithm utilizes a fully precomputed lookup table stored in memory to calculate the results in $\mathbb{F}_{2^8}$.
- *LogTable*$\{8, 16\}$: This algorithm uses a reduced lookup table to calculate the results in $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{16}}$. The log table minimize memory consumption at the cost of additional operations for every calculation.
- *ExtendedLogTable*$\{8, 16\}$: This algorithm extends the lookup table used by the *LogTable* to calculate the results in $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{16}}$. The extended lookup table removes a number of checks necessary in the *LogTable* algorithm when moving from exponential to polynomial representation.

The following figures show the throughput for the arithmetic operations tested. The benchmark uses two generations each containing g packets where each packet is 1400 B long. From each generation two packets are then randomly selected and the specified operation is performed. In the operations tested a constant is used for the multiplication, this constant was randomly generated for each invocation of the operations under test. For each operation, a number of iterations were completed so that the total measurement time exceeded a minimum of 10 ms, this was done to keep inaccuracies due to timing granularity and other disturbances in the measurements low. For each operation this was repeated 100 times.

The benchmarks were run on the device specified in Table I using two different generation sizes: $g = 32$ and $g = 1024$. These numbers were chosen to see how the algorithms were affected by the working set size (which is the generation size multiplied by the packet size). The working set size can have a significant impact on performance due to caching effects [14, p. 593-673]. In this case we only observe a slight drop in performance as the working set size increases indicating that the CPU is able to keep the working set in the cache.

All implementations presented here are written in C++ using no assembler or compiler intrinsics to further speed up the computations. However, inspecting the assembly output of the compiled benchmark does reveal that the compiler was able to take advantage of vectorized Single Instruction Multiple Data (SIMD) instructions in some of the arithmetic loops. Although this boosts performance considerably some functions were not

optimized by the compiler. It is therefore likely that further performance gains could be achieved by hand-writing some operations using assembly or vectorized SIMD instructions.

In Figure 2 we see the benchmarks for a generation size of 32. In this case we see that the OPF preforms better than the alternative implementations. One interesting thing to observe is that the OPF performs slightly faster in subtraction than addition. The explanation for this is that addition requires two checks, one for checking for integer overflow, and one for checking whether the prime modulo operation must be performed. For subtraction we only have to check for integer underflow. Avoiding this additional check yields an approx. 5% performance increase.

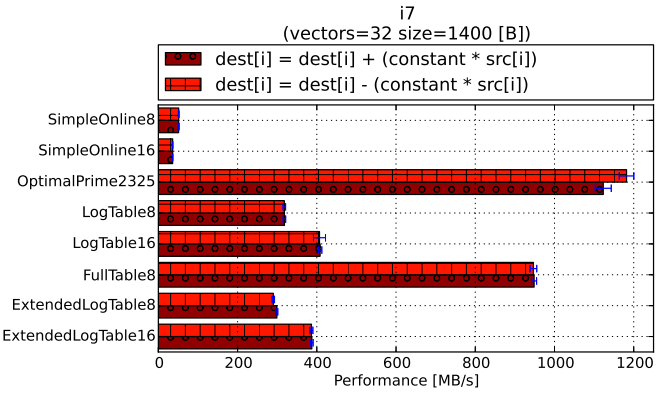


Fig. 2: Throughput for the compound NC operations for a generation size of 32.

In Figure 3 we see the benchmarks for a generation size of 1024. Where the tendency remains the same as for the generation size of 32.

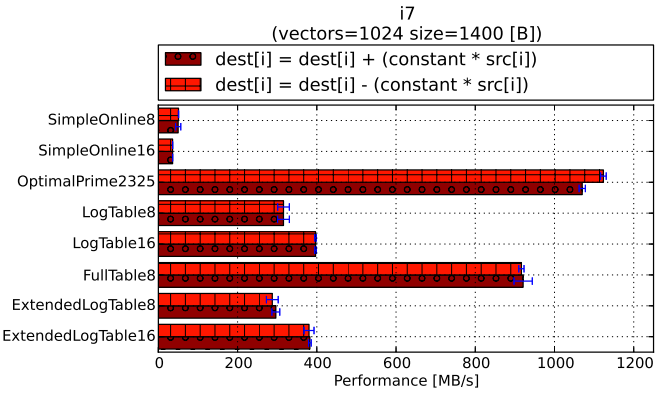


Fig. 3: Throughput for the compound NC operations for a generation size of 1024.

As seen the two implementations `FullTable8` and `OptimalPrime2325` are by far the fastest. Where the `OptimalPrime2325` on average provides an 18% performance increase for addition and an average 23% performance increase for subtraction when compared to the `FullTable8`.

VIII. Conclusion

In this paper we present our initial investigation of Optimal Prime Field (OPF) for Network Coding applications. The results show that OPF looks like a promising addition to the selection of finite field implementations. Besides good performance one of the main benefits is the large field size and the limited memory consumption required by the algorithms. As mentioned the OPF implementation uses pure C++ code, and inspecting the compiled assembly we could verify that the compiler did not optimize all operations using SIMD instructions. It is therefore likely that even better performance can be obtained. The main drawback of this approach is the need for the prefix binary mapping scheme. Further work should be invested in reducing the overhead added by this. We expect OPFs to be particularly useful on small embedded devices, with only KB's of memory, since these devices cannot use the lookup table based algorithms.

Acknowledgments

This work was partially financed by the CONE project (Grant No. 09-066549/FTP), the Colorcast project (Grant No. 12-126424/FTP) both granted by Danish Ministry of Science, Technology and Innovation as well as by the collaboration with Renesas Mobile throughout the NOCE project.

References

[1] R. Ahlswede, N. Cai, S. Y. R. Li, and R. W. Yeung, "Network information flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, 2000.

[2] S. Lin and D. J. Costello, *Error Control Coding, Second Edition*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2004.

[3] K. M. Greenan, E. L. Miller, and T. J. E. Schwarz, "Optimizing galois field arithmetic for diverse processor architectures and applications," in *MASCOTS*, E. L. Miller and C. L. Williamson, Eds. IEEE Computer Society, 2008, pp. 257–266.

[4] C. Fragouli and E. Soljanin, "Network Coding Fundamentals," *Foundations and Trends in Networking*, vol. Vol. 2, Issue 1, pp. 1–133, 2007.

[5] J. Heide, M. V. Pedersen, F. Fitzek, and T. Larsen, "Network coding for mobile devices - systematic binary random rateless codes," in *Workshop on Cooperative Mobile Networks 2009 - ICC09*. IEEE, Jun. 2009.

[6] J. Heide, M. V. Pedersen, F. H. Fitzek, and M. Médard, "On code parameters and coding vector representation for practical rlnc," in *IEEE International Conference on Communications (ICC) - Communication Theory Symposium*, Kyoto, Japan, 5-9 June 2011.

[7] P. Crowley. (2006, Nov.) Gf(232-5). [Online]. Available: http://www.lshift.net/blog/2006/11/29/gf232-5

[8] D. V. Bailey and C. Paar, "Optimal extension fields for fast arithmetic in public-key algorithms," in *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*. London, UK: Springer-Verlag, 1998, pp. 472–485. [Online]. Available: http://portal.acm.org/citation.cfm?id=646763.706317

[9] S. B. Mohan and B. S. Adiga, "Fast algorithms for implementing rsa public key cryptosystem," *Electronics Letters*, vol. 21, 1985.

[10] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[11] T. Granlund. (2011, Mar.) Instruction latencies and throughput for amd and intel x86 processors. [Online]. Available: http://gmplib.org/tege/x86-timing.pdf

[12] Steinwurf ApS. (2012) Kodo git repository on github. [Online]. Available: http://github.com/steinwurf/kodo

[13] ——. (2012) Fifi git repository on github. [Online]. Available: http://github.com/steinwurf/fifi

[14] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 2nd ed. USA: Addison-Wesley Publishing Company, 2010.